SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

PHANtom: a Modern Aspect Language for Pharo Smalltalk*

Johan Fabry*, Daniel Galdames

PLEIAD Laboratory, Computer Science Department (DCC), University of Chile – Santiago, Chile,
http://pleiad.cl

SUMMARY

In the context of our research on Aspect-Oriented Programming, we have a need for a modern and powerful aspect language for Smalltalk. Current aspect languages for Smalltalk however fall short on various points. To address this deficit, we elected to design and build PHANtom: a modern aspect language for Pharo Smalltalk. PHANtom is designed to be an aspect language in the spirit of Smalltalk: dynamic, simple and powerful. PHANtom is a modern aspect language because it incorporates the best features of languages that precede it, includes recent research results in aspect interactions and reentrancy control, and is designed from the onset to be optimized and compiled where possible. In this paper we present the latest version of the language and give examples and patterns of use. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: PHANtom, Aspect-Oriented Programming,Smalltalk

1. INTRODUCTION

To address the issue of code for one concern being scattered among different classes in an application, Aspect-Oriented Programming (AOP) proposes to modularize such cross-cutting concerns into a new kind of module: an *aspect*. Aspects are a different kind of module because they not only implement the behavior of a concern, but also specify when this behavior should be invoked. To allow this, conceptually each step of the application is reified in what is called a *join point*. The execution of the application consequently produces a stream of join points that is presented to the aspects. The aspects identify relevant join points, *i.e.,* steps in the execution of the application, by means of *pointcuts* that match on the stream of join points. The behavior of the aspect, called *advice*, is linked to the pointcut such that when a pointcut matches, the corresponding advice is executed. To make this all possible, a special tool called the *aspect weaver* implements all the required infrastructure. One possible implementation strategy is that the aspect weaver is a compiler. This compiler modifies each source code location that produces join points of interest to the aspects, locations known as *join point shadows*. At each of these the modified code then executes the required advice, as specified by the aspects.

A significant part of the research we perform is in the domain of AOP, where we are interested in interactions between aspects [5], as well as Domain-Specific Aspect Languages (DSALs) [6]. For our latest experiments with AOP we chose to use Pharo Smalltalk, which meant we also required an aspect language that incorporates recent AOP research results, is solid, is powerful

*This is an extended version of the paper with the same name published at IWST2011 [9]
*Correspondence to: DCC-Universidad de Chile, Blanco Encalada 2120, Santiago, Chile, http://pleiad.cl

Contract/grant sponsor: Johan Fabry is partially funded by FONDECYT; contract/grant number: 1090083

and is extensible. When evaluating existing aspect languages for Smalltalk we however found these did not meet our criteria and hence set out to build our own language, called PHANtom.

PHANtom is designed to be an aspect language in the spirit of Smalltalk: dynamic, simple and powerful. PHANtom is a modern aspect language because it incorporates the best features of languages that precede it, includes recent research results in aspect interactions and reentrancy control, and is designed from the onset to be optimized and compiled where possible. PHANtom adds to the state of the art in aspect languages as its joining of recent research results is new. Moreover, it improves on existing aspect languages in Smalltalk on various points, *e.g.* by allowing for advice execution ordering, as is discussed later. We therefore consider that PHANtom is also of interest to the Smalltalk programmer wanting to use AOP.

In this paper we present the PHANtom language, along with some relevant parts of its current implementation.

The structure of the paper is as follows: in Section 2 the language is introduced, starting with the core interplay between join points, pointcuts, advice, inter-type declarations and aspects in Section 2.1. This is followed by a presentation of the advice ordering features, in Section 2.2, and control of reentrancy in Section 2.3. Section 3 shows common usage patterns of PHANtom and Section 4 details two refactorings of existing software to use PHANtom. This is followed by a discussion of related work in Section 5, before the paper concludes and presents future work in Section 6.

2. THE PHANTOM LANGUAGE

PHANtom is designed to be an aspect language fully in the spirit of dynamic languages like Smalltalk. This is achieved by combining dynamism, simplicity and power. Firstly, PHANtom is dynamic with regard to changes in the code base: it allows for classes and aspects to be added, removed and changed at runtime. PHANtom is also dynamic in the sense of "dynamic AOP" which means that aspects can be added to and removed from the system at runtime, through the process of deployment and undeployment. Secondly, we aim for simplicity by having a minimal set of language constructs who are as free from usage restrictions as possible. Moreover each of these has a straightforward and uniform behavior. Thirdly all language constructs are first class objects, thus providing all the power of first-class constructs. Additionally, PHANtom provides aspect language features, absent in other aspect languages, that give programmers more power over the order of aspect execution and more powerful reentrancy control.

We call PHANtom a modern aspect language for three reasons. First, it was designed taking into account existing aspect languages. It incorporates what we believe to be the best features of these languages. Second, it incorporates recent important research results that have not yet been used together. Last but not least, it was designed with compilation and optimization in mind. While currently PHANtom is an interpreted language, key future work is to compile aspects.

A number of existing aspect languages have been influential in the design of PHANtom. We mention these influences here but refer to related work for a description of these languages. From AspectJ [13] we have taken the use of patterns in pointcuts and advice ordering through precedence declarations. AspectS [12] inspired our dynamic AOP features, the use of method wrappers [4], and having all constructs available as first class objects. Eos-U [16] introduced what is known as a more symmetric view of AOP, which is also present in PHANtom: aspects are as classes in that they can be instantiated, their behavior is present in methods, and aspects can match join points of other aspects.

The contributions of PHANtom to the aspect language research domain is the specific combination of the advanced language features of advice ordering (see Section 2.2) and reentrancy control (see Section 2.3). Also, to the best of our knowledge, this is the first time a granularity refinement has been proposed to the static AspectJ aspect ordering scheme (see Section 2.2). Beyond these contributions and considering aspect languages in Smalltalk, to the

DOI: 10.1002/spe

best of our knowledge, PHANtom is the only aspect language for Smalltalk that provides for reentrancy control and was designed with optimization in mind.

Currently, no special syntax has been developed for PHANtom. Instead, aspects are built by constructing their component parts using standard Smalltalk, *i.e.,* by instantiating the corresponding Smalltalk objects and adding them to the aspect as needed. In this we follow the Smalltalk philosophy that everything is realized by sending messages to objects, even the creation of classes (which is the sending of the subclass: message to a class).

2.1. Overview of the Core of PHANtom

We now proceed with giving an overview of core of PHANtom, using some illustrative examples. We show here the interplay between pointcuts, advice, inter-type declarations and aspects by means of a small example aspect described below. In the next sections we detail more advanced capabilities of PHANtom: advice ordering (Section 2.2) and reentrancy control (Section 2.3).

As an example initial aspect, in this section we build an extension to the SUnit test suite that counts the number of assertions that are performed in the execution of a given collection of test classes. Note that, due to the simplicity of the example, alternative implementations may be envisioned that provide the same behavior. As it is not the goal of this paper to argue for AOP, but to introduce PHANtom, we have chosen to keep the example plain to ease readability of this text.

2.1.1. Join Point Model In AOP, the join point model of an aspect language defines the events that are produced by the application when it runs, and their properties. Each such event is a join point, and an aspect defines at which join points its behavior is executed.

Following the goal of simplicity, and in accordance with the Smalltalk philosophy that a computation consists of objects and messages, the join point model of PHANtom consists solely of method executions. Also, each join point contains a reference to the sender, the receiver and the arguments of the message that lead to the method being executed.

2.1.2. Pointcuts Pointcuts are what is also known as the quantification part of the aspect: a pointcut is responsible for determining when the behavior of an aspect is triggered. More precisely, pointcuts are predicates over the stream of join points that is emitted by the base application as it runs. When pointcut matches a join point, the advice that is linked to that pointcut executes.

Basic Pointcut Definitions In PHANtom, the basic pointcuts use static information of the join point to decide whether they match, *i.e.,* pointcuts filter the join point stream based on the type of the receiver and the selector name. Such pointcuts are instances of the PhPointcut class and an example instantiation of a pointcut is below:

```
PhPointcut receivers: 'TestCase' selectors: 'assert:'.
```

The above code matches all executions of the method with selector assert: where the receiver of the method is an instance of TestCase (excluding its subclasses).

Type	Pattern	Meaning	Type	Pattern	Meaning
Receiver	+	Class and Subclasses	Selector	#()	All Selectors
	*	String Match		_	A Keyword of Selector

Table I. Patterns and their meaning.

PHANtom allows AspectJ-like patterns to be used in the string that specifies a receiver class name or selectors, as outlined in Table I. Allowed patterns for receiver class are as in AspectJ. They allow for the use of two special symbols: a + which is used as a suffix to a class name,

Prepared using speauth.cls
DOI: 10.1002/spe

and a * which is used anywhere within the class name. The + indicates that the receiver of the method should be an instance of the indicated class, or an instance of a subclass. The * is the string match like in regular expressions, except that it does not automatically match on metaclasses. For example, a pattern Test* does **not** match TestFoo class. To match instances of metaclass, the receivers pattern must include the word class. In the example, a match would instead be made when using the pattern Test* class.

For selector patterns, an array of size zero: #(), matches all selectors and the wildcard _ may also be used[‡]. This wildcard matches unary and binary messages, as well as one keyword in a keyword message. In a keyword message with multiple keywords, _: matches one keyword and may be used more than once. For example, with:_:with:_: matches with:for:with:and:, with:with:with:with:, and so on.

Using the patterns we now start on the pointcut that will be used in the example assertion count aspect. Let us assume we wish to count all assertions,which can be either assert:, assert:equals:, assert:description: or assert:description:resumable:. However, we only wish to do this for a select group of test case classes, namely all subclasses of ACTestCase. The pointcut that matches those join points is given below:

```
PhPointcut receivers: 'ACTestCase+' selectors: #(assert: assert:_: assert:_:_:)
```

This code shows the use of the + symbol and the ability to use an array of selectors. The use of patterns has been described above. Using arrays of selectors allows us to define a pointcut that matches on different selectors. The receivers may also be specified as an array, where each element in the array is a class name pattern.

Context Exposure Context exposure is the means by which information that is present in the join point is made accessible to the advice. Declaring this in the pointcut allows the aspect weaver to perform optimizations: avoiding creation and/or reification of elements that are not used in the advice [11]. If context exposure is specified in a pointcut, the corresponding advice will be passed an instance of PhContext that contains the specified context values. These can be as follows:

- receiver includes the object that received the message
- sender includes the object that sent the message
- selector includes the selector of the message
- arguments includes the arguments of the message
- proceed allows for the original behavior to be invoked
- advice allows for advice execution order to be modified

Most of the above context values are straightforward. An example of the use of proceed will be given in Section 2.1.3 and an example of using advice is present in Section 2.2.2. Below we show how to add context exposure to the above pointcut, specifically including the receiver object in the context.

```
PhPointcut receivers: 'ACTestCase+'
           selectors: #(assert: assert:_: assert:_:_:)
           context: #(receiver)
```

Package Restrictions In the above example, for assertions to be counted the test case class needs to be a subclass of ACTestCase. Usually, test case classes are however subclass of TestCase. We therefore propose an alternative approach for our running example aspect, which is using package restrictions. This alternative is to consider all assertions in test classes that belong to a given category, *e.g.* named Tests-Mine. We achieve it using a package restriction.

[‡]as * is a valid selector.

DOI: 10.1002/spe

Such a restrict: specification restricts the matching of classes to those which are present in the array of specified categories. An example is shown below. It yields the pointcut that we will for our example aspect. Hence we store it in a temporary variable for later use.

```
pc := PhPointcut receivers: 'TestCase+'
                 selectors: #(assert: assert:_: assert:_:_:)
                 context: #(receiver)
                 restrict: #(Tests–Mine).
```

Pointcut Combinations and Custom Pointcut Matching Pointcuts can also be combined using logical and, or and not operators. PhPointcut instances understand the corresponding messages &, |, not. Sending a not returns the negation of the receiver pointcut. The former two perform, resp. the logical and, or, taking a second pointcut as argument. The context exposure of the resulting pointcut is the union of the context exposures of both pointcuts.

To provide more power for pointcut specification, PHANtom allows for the programmer to define custom pointcut patterns if required. This is made possible by using the PetitParser parser generator framework [17] in the implementation of the matching of pointcut patterns. Instead of using a string or array as argument of the receivers or selectors specification, a PetitParser parser can be passed as an argument for either, or both. This parser is given the complete class description of candidate classes, or candidate selector strings, respectively. If the parser matches, the class is a valid receiver or selector. A complete discussion of PetitParser parsers is out of the scope of this paper. We only provide the following example that uses two custom parsers. It matches the count and count: messages of instances of classes that have one instance variable named count.

```
PhPointcut receivers: (#any asParser plusGreedy:  'instanceVariableNames: ''count''' asParser)
           selectors:  ('count' asParser , ':' asParser optional).
```

Dynamic Pointcuts The basic pointcuts, as defined above, serve as a producers of dynamic pointcuts: sending them either a testing or control flow message produces a new pointcut that incorporates the provided guard or control flow check, respectively. We now discuss both of these messages in more detail.

Guarded pointcuts are pointcuts that contain a runtime condition. They are produced by sending an if: message to an existing pointcut. At runtime, for every join point that statically matches this receiver pointcut a test is performed to determine whether the guarded pointcut matches. The test to perform is the block given as argument to the if: message. When run, the block is passed one argument: the context as exposed by the receiver pointcut. If the block returns true, the pointcut matches, if it returns false, the pointcut does not match.

As an example, we provide an alternative implementation of the pointcut pc defined above. We obtain a package restriction by dynamically checking that the package name of the message sender is equal to Tests-Mine. Note that this pointcut produces a different type of overhead than the pc pointcut. The package restriction of the former is evaluated once, when the pointcut is installed. In contrast, the sender category verification of the latter is evaluated every time a joinpoint is matched by the pointcut that receives the if: message.

```
(PhPointcut receivers: 'TestCase+'
            selectors: #(assert: assert:_: assert:_:_:)
            context: #(receiver sender))
    if: [:ctx| ctx receiver category = #Tests–Mine].
```

To dynamically verify if one join point occurs within the control flow of a second join point, the inCflowOf: message is used. It returns a pointcut that performs a control flow check at runtime. This pointcut will match at join points where first the reciever pointcut statically matches and second the argument pointcut matches a join point that is part of the current

DOI: 10.1002/spe

control flow. The antonym of inCflowOf: is notInCflowOf:, returning a pointcut that matches the occurrence of the receiver that is not in the control flow of the argument.

As an example, we refine the pc pointcut we defined above to take into account the implementation of assert:equals:. This method calls the assert: method as part of its implementation. As a result, when using the pc pointcut, both the assert:equals: and the assert: method execution will be counted whenever a call to assert:equals: is performed. However in these cases we do not want to count the assert: method execution, as it does not correspond to an additional assertion being performed. To achieve this, we can use a control flow pointcut, as implemented below:

```
assert := (PhPointcut receivers: 'TestCase+' selectors: #assert:
                    context: #(receiver) restrict: #(Tests–Mine))
                                notInCflowOf: pc.
allasserts := assert | (PhPointcut receivers: 'TestCase+' selectors: #(assert:_: assert:_:_:)
                                context: #(receiver) restrict: #(Tests–Mine))
```

The first statement of the example only matches calls to the assert: method execution that do not happen when assert:equals: (or any other method that matches the pc pointcut) is executing. This pointcut is used in the second statement to build a pointcut that matches on the assertions we wish to count by also matching on the other variants of the assert message.

2.1.3. Advice Next to the quantification part of the aspect, the pointcut, the advice determines the behavior to be executed when a pointcut matches. In PHANtom, we choose to have the behavior implemented as regular methods, inspired from Eos-U [16]. Advice, which are instances of PhAdvice, merely associate a pointcut to the execution of this behavior. This is different from Eos-U, which does not have the notion of advice. It instead places responsibility of determining what method to run on the pointcut itself. This however restricts reuse of the pointcut, as it is now tightly coupled to a specific method name for the advice. Furthermore, the interplay of specifying a method in the pointcuts with combining two pointcuts (with &, |) is not immediately obvious. An argument can be made for various different choices of when and how the behavior of one of the constituent pointcuts should be executed when the composed pointcut matches. This however adds complexity to the language, which we aim to avoid.

Creation of a PhAdvice takes as arguments the pointcut to be used, the name of a message to send and its receiver, and when this message must be sent. The sent message will be given one argument: the join point context as exposed by the pointcut. Note that, for the sake of brevity, in the remainder of this text we will use the term *executing the advice* to mean the execution of the method that results from the message sent by the PhAdvice instance when the pointcut matches. Also, in this section we consider a situation where only one advice needs to execute at a given join point. The case where multiple advice execute at one join point is discussed in detail in Section. 2.2.

Below is the advice for our assertion count example:

```
adv := PhAdvice before: allasserts send: #incCount: to: self
```

This code specifies that we wish that before the execution of matches of the pointcut allasserts (which we declared in Section 2.1.2) the incCount: message is sent to self. The implementation of this message will be discussed in Section 2.1.5.

There are three types of advice: *before*, *after* and *around* advice, each with their respective keyword: before:, after: and around:. Before advice executes before the join point, *i.e.,* before the selected method execution, after advice executes after the join point and returns the value of the join point execution. Around advice executes instead of the selected join point, effectively replacing the original behavior of the method with the behavior of the advice. When around advice executes it is however possible to invoke the original behavior of the method by sending the proceed message to the context. The original behavior can be executed with different arguments, by using the proceed: message and passing an array of new arguments. Note that proceed and proceed: may be sent any number of times.

DOI: 10.1002/spe

Strictly speaking, before and after advice are redundant. This is as the behavior of the former can be obtained by using an around advice that ends with a proceed, and the latter by an around advice that starts with a proceed (returning the result of the proceed send at the end of the advice). The use of before and after advice however allows the aspect weaver to perform optimizations, as they do not require the original behavior, nor receiver and arguments to be reified into the context.

To allow for more flexibility and conciseness, and inspired from AspectS, instead of a message send specification a block may also be used by a PhAdvice. The block takes as argument the context, and instead of sending a message, advice execution consists in evaluating the block. For example, the code below will print the receiver on the transcript after the assertion methods execute.

```
PhAdvice after: allasserts advice: [:ctx | Transcript show: (ctx receiver asString); cr.].
```

Note that this feature is mainly intended as a facility for fast prototyping or simple aspects. Its use at a larger scale, for more complex aspects, is discouraged as it does not combine well with advanced features such as dynamic advice ordering (see Section 2.2).

2.1.4. Inter-Type Declarations Inter-type declarations, also known as introductions or structural aspects, are a means to perform modifications of classes such that they contain the desired state and behavior required for the functionality of the aspect. In aspect languages for statically typed languages, *e.g.* AspectJ, these modifications may also include modifications to the type hierarchy to obtain the required behavior.

In PHANtom, inter-type declarations are purely a means to permit modular class extensions. They are instances of PhClassModifier and they specify additions of variables and methods. This may be performed both at instance and at class side of the target class. The target of the addition is specified with a static pointcut, *i.e.,* pointcuts generated by a inCflowOf: or if: and their combinations are ignored. The resulting targets are all classes whose method executions are (statically) matched by the pointcut.

The motivation for this class extension mechanism in addition to the standard class extensions in Pharo is threefold. First, inter-type declarations allow the addition of variables, whereas standard class extensions do not allow variables to be added. Second, their application is specified using a pointcut, which allows one extension to be added to multiple classes simultaneously. This in contrast to class extensions, which are strictly one-to-one. Third, the inter-type declarations can be textually contained within the definition of the aspect. Arguably this is the module where they should be placed. In contrast, standard class extensions place their definition in the target class, which then also appears in the same category of the aspect.

For our running example, we need the test cases to contain a variable that maintains the number of times that assertions have been called. For it to be available to the aspect we also require accessor methods that expose this variable. This is realized by the below two inter-type declarations:

```
cm1 := PhClassModifier on: pc addIV: 'phacount'
cm2 := PhClassModifier on: pc addIM:
   'phacount
      ↑phacount ifNil: [phacount := 0]'.
cm3 := PhClassModifier on: pc addIM:
   'phacount: anObject
      phacount := anObject'.
```

The first PhClassModifier instance declares the addition of a phacount instance variable to classes matched by the pc pointcut, *i.e.,* the subclasses of TestCase that are in the category named Tests-Mine. The second instance adds the corresponding accessor to these classes and the third declares the corresponding mutator.

When the aspect is deployed (see Section 2.1.5), the inter-type method variable additions will be added to the target class, which is then recompiled. If the class already contains this

Prepared using speauth.cls
DOI: 10.1002/spe

variable, an exception will be thrown. Similarly, method declarations will be compiled within the scope of the target class and added there. If methods with the same selector are already present in the class, an exception will be thrown.

To add class variables, resp. class methods, the PhClassModifier needs to be instantiated with the on:addCV:, resp. on:addCM: messages.

2.1.5. Aspects and their Deployment An aspect is a class that modularizes cross-cutting behavior, by means of pointcuts, advice and inter-type declarations. In PHANtom, aspects are subclasses of the PhAspect class, and contain a collection of PHAdvice and PHClassModifiers (and each PHAdvice keeps a reference to a PhPointcut instance). Instances of aspects are not automatically active. To add the inter-type declarations and have advice execute when their pointcuts match, aspects must be deployed. Conversely, a deployed aspect can be undeployed, which removes its effects on the system.

To complete our running example, we first need to subclass PhAspect and add the initialize method as below,

```
initialize
  |pc assert allasserts adv cm1 cm2 cm3|
  super initialize.
  "... take pc, asserts, allasserts, adv, cm1, cm2, cm3 from above ..."

  self add: adv.
  self addClassModifier: cm1.
  self addClassModifier: cm2.
  self addClassModifier: cm3.
  self install.
```

Adding the advice to self achieves that the incCount: message is sent to the aspect after all executions of the assert methods identified by the pointcut of the advice. Adding the three class modifiers ensures that ACTestCase understands the phacount and phacount: messages and behaves accordingly.

All that remains to be done is to define the behavior of incCount:, which is done by adding the below method to our aspect:

```
incCount: aContext
   aContext receiver phacount: aContext receiver phacount + 1
```

The implementation of incCount: uses the context to obtain the receiver (an instance of ACTestCase or a subclass thereof) and simply increments the count by one.

Using the aspect is straightforward, as instantiating sends it the install message that deploys it in the system. This immediately makes all ACTestCase and subclass instances count their assertions. The assertion count of each can be obtained through the accessor method when needed. Sending the uninstall message to the aspect undeploys it[§].

Note that the aspect weaver maintains a list of all installed aspects, which is obtained as follows: PhAspectWeaver installedAspects. This can be useful to uninstall aspects whose references are out of scope.

2.2. Advice Ordering

In the above discussion we assumed that at a given join point only one advice will execute. In this section we detail the behavior of PHANtom when various advice need to execute at one given join point. We now first specify the default behavior, before considering the need for program-specific behavior and detailing the two advice ordering features offered by PHANtom.

[§]Although we speak of deployment and undeployment of aspects, these messages are called install: and uninstall, which was how they were originally named in AspectS [12]

DOI: 10.1002/spe

By default, if multiple advice specify that they should be run at a given join point, they will be run in sequence. The entire process is illustrated in Figure 1. First all before advice are run in an unspecified order. Then an around advice is run (if any need to be run). If this advice sends a proceed message, instead of executing the original behavior, another around advice is run (if any more need to be run), and so on. The order in which the around advice are selected is unspecified as well. Last, all after advice are run, again in an unspecified order.

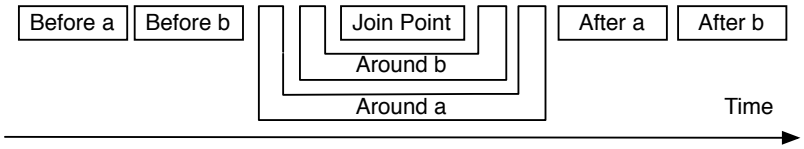

Figure 1. Execution order of multiple advice for one join point

An important issue when multiple aspects, or multiple advice, are present is however their order of execution as it can significantly impact the behavior of the application. Aspect execution order is one case of interaction between aspects [5]. A typical example is the interaction between a timing and a logging aspect, which is as follows: Timing adds around advice to a number of methods, sending a proceed message and registering the time it needs to execute. Logging adds around advice to a number of methods, sending a proceed message and logging it. When both aspects affect the same method, the situation is similar to the around advice execution depicted in Figure 1. Timing will or will not include the time taken for the log operations, depending on which advice executes first, *i.e.,* whether Around a is the timing advice and Around b is the logging advice or vice-versa. The required behavior is clearly a decision that should be taken when developing the application, and hence the aspect language should provide for a means to define an ordering for the execution of advice.

PHANtom has two mechanisms to define the ordering of advice. It first has a precedence relationship inspired from AspectJ, which we discuss next. Second it has a fully dynamic advice order list that can be manipulated by the advice themselves, which is discussed afterward.

2.2.1. Deployment Ordering The first aspect ordering scheme provided by PHANtom is a refinement of the static aspect precedence declaration scheme of AspectJ, adapted to a setting with aspect deployment. A precedence declaration determines that an aspect, or a set of aspects, is more important than another (set of) aspect(s). PHANtom implements the global precedence scheme of AspectJ and adds to it a local precedence scheme. In Table II, we give an overview of both precedence schemes, and we talk about them next.

Receiver	Message	Ordering
aPhAspect	precedence:	Add precedence declarations to the global order.
aPhPointcut	precedence:	Add local order to the global order, ignore conflicting global precedences.
aPhPointcut	overridePrecedence:	Ignore all precedences of the global order, only use the local order.

Table II. Deployment ordering specification API.

To declare a global precedence order, the precedence: message is sent to a PhAspect instance, with as argument an ordered collection of receiver class patterns (see Section 2.1.2). The first element of the collection determines the most important aspect(s), the second element determines (a) lesser important aspect(s), and so on.

DOI: 10.1002/spe

Receiver	Message	Result
aPhContext	beforeAdvice, beforeAdvice:	Obtain, set sequence of before advice
aPhContext	aroundAdvice, aroundAdvice:	Obtain, set sequence of around advice
aPhContext	afterAdvice, afterAdvice:	Obtain, set sequence of after advice
aPhContext	currentAdvice	Obtain sequence of advice that is currently executing
aPhContext	currentAdvice:continueAt:	Set currently executing sequence and index of where to continue

Table III. Dynamic ordering specification API.

The group of all declared aspect precedence relationships should form a partially ordered set of aspects, which allows a directed acyclic graph of aspect dependencies to be formed. When a declared precedence does not respect the partial order, a cycle would be formed in the dependency graph. At aspect deployment time the graph is therefore verified for cycles. If any are present, an error is produced and the aspect is not deployed.

The aspect dependency graph is used at advice execution time to order the advice execution as in AspectJ, which is as follows: The order of before advice and around advice is from more important to less important, *i.e.,* executing the advice of the most important aspect first. The order of execution of after advice is however the reverse: from less important to more important, *i.e.,* executing the advice of the most important aspect last.

Fine-Grained Deployment Ordering In addition to declaring a global precedence order, a precedence order with a more fine-grained scope can be declared, if necessary. To the best of our knowledge, this is the first time such a refinement has been proposed to the AspectJ aspect ordering scheme. The finer-grained ordering is performed by sending the precedence: message to a pointcut. As a result, the precedence specification that is passed as argument is present in the join points that match the pointcut. In other words, in those join points the precedence order is the global order that is extended with the order declared on the pointcut. The order declared on the pointcut is considered to be more important than the global order: If joining both orders leads to cycles in the dependency graph, the precedence declarations of the aspect that cause the cycle are ignored.

Lastly, PHANtom also allows for the precedence of a pointcut to override the global precedence, by sending the overridePrecedence: message to a pointcut. This sets the precedence order for the matched join points to be exclusively the order that is declared in the pointcut. If this order contains conflicts, an error is produced at aspect deployment time

2.2.2. Dynamic Ordering Combining dynamism and power, PHANtom also allows for aspect ordering to be decided when advice is executing. This is inspired by the work on Dynamic AspectJ by Assaf and Noyé [1]. In this work, the authors argue for a more dynamic approach to the topic of multiple advice execution. They propose an extension to AspectJ that allows advice execution to be scheduled dynamically, including the possibility for some advice execution to be skipped. At runtime, each advice can obtain a representation of the list of advice that is scheduled to be executed at this join point, which is called an *AspectGroup*. Operations on the AspectGroup allow execution of all advice to be skipped, execution of advice with a certain name to be skipped, and positioning advice with a given name at the head of the sequence. By repeatedly using the latter operation the sequence can then be reordered.

In accordance to the aim of simplicity, the API of PHANtom is conceptually more straightforward and less restrictive than Dynamic AspectJ, and is shown in Table III. In PHANtom, a pointcut may expose the advice ordering in the context through the advice keyword. The context can then be queried for the list of scheduled before, after and around advice, as well as for the currently executing sequence of advice. This returns an ordered

DOI: 10.1002/spe

collection of such advice. This collection can be manipulated as required by the advice, and the context can be given a new sequence of scheduled advice, setting the execution sequence. When changing the group of scheduled advice to which this advice belongs, it is however ambiguous where in the sequence the advice execution process should continue. To avoid this, such a change must be performed by sending the currentAdvice continueAt: message. This message also indicates at what index in the sequence is the advice that should be run next, removing the ambiguity. The other advice setter messages (beforeAdvice: when used in a before advice, ...) are silently ignored.

2.3. Reentrancy Control

A common pitfall in the use of aspects is the appearance of infinite loops when aspects are added to the application. Typically what occurs is that the pointcut of an advice captures the behavior of the advice itself. An example of this is shown in Figure 2. When the advice executes, its pointcut matches its own execution. As a result, the advice executes and its pointcut matches its own execution. Consequently, the advice executes, and so on.

```
|asp|
asp := PhAspect new.
asp add: (PhAdvice before: (PhPointcut receivers: 'Transcript class' selectors: 'show:')
              advice: [ Transcript show: 'Showing ' ]).
asp install.
Transcript show: 'reentrancy control'; cr.
```

Figure 2. A potential infinite loop as part of the advice execution is captured by its own pointcut.

In this section we introduce the reentrancy controls provided by PHANtom. Based on the concepts of computational membranes, PHANtom provides a safe default, that can however be modified by the programmer if required.

2.3.1. Computational Membranes for Reentrancy Control A number of proposals have previously been made that aim to avoid such reentrancy issues. The recent work by Tanter on execution levels for aspect-oriented programming [18] provides a thorough analysis of the problem, and a solution that supersedes previous proposals. The latest work of Tanter *et al.*, *computational membranes* [19], is a generalization of execution levels that provides more flexibility. PHANtom uses these computational membranes to provide for reentrancy control.

At its core, the work on membranes proposes to deploy membranes around a given computation, to serve as a scoping mechanism for the join points emitted by that computation. Membranes are then used to structure execution of aspects. Membranes can be nested, resulting in a hierarchy of membranes, and crosscut, when multiple membranes are deployed around (a part of) the same computation. Figure 3 (taken from [19], with permission) illustrates how membranes can be deployed on a control flow of three computations X, Y and Z. Membranes m1 and m2 crosscut, as they both wrap the Y computation.

Membranes scope the observation of join points by controlling the propagation of join points of the computation on which they are deployed. For an aspect to observe a join point, it needs to register itself in a membrane. In Figure 3, the aspect A is registered in the membrane m3. All the join points that are observed by that membrane are visible to the aspect, *i.e.,* can be matched by its pointcuts. To observe a join point, a membrane however must first be bound to a membrane that generates join points (which may be itself). In the example, the membrane m3 is bound to membrane m1. As a result, the join points generated in membrane m1 'flow' to the membrane m3 and hence are visible to the aspect A. Note that join points that occur

DOI: 10.1002/spe

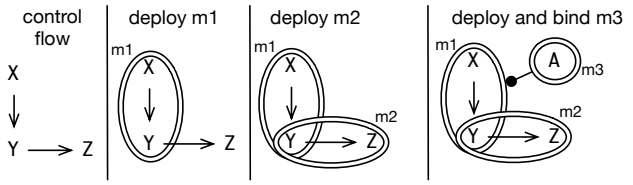

Figure 3. Deployment of membranes (taken from [19], with permission). Membranes m1 resp. m2 wrap computation X and Y resp. Y and Z. Membrane m3 is bound to m1 causing join points from the computation in m1 to be visible to the aspect A.

Receiver	Message	Result
aPhMembrane	advise:, unAdvise:	Binds, resp. unbinds the argument membrane to the receiver.
aPhMembrane	registerAspect:, unregisterAspect:	Register, resp. unregister an aspect instance in the receiver.
aPhMembrane	pointcut:	Pointcut argument defines computation around which the receiver will be deployed.
aPhMembrane	install, uninstall	Deploy, resp. undeploy the receiver.

Table IV. The Membranes API

in the computation Z will not be seen by the aspect A, as the membrane m1 does not capture this computation.

Note that membranes are more powerful than what is presented here, and PHANtom only implements the subset of the capabilities of membranes that allows for topological scoping of join points. For example, the membranes proposal posits that membranes may filter the incoming and outgoing join points. For more information we refer to the membranes publication [19].

2.3.2. Addressing Reentrancy Issues Following the goal of simplicity, PHANtom provides straightforward behavior: it by default avoids the typical reentrancy issues we identified above. Recall that in Figure 2 the advice sends the show: message to the Transcript, which is captured by its own pointcut¶. Without reentrancy control, it hence leads to an infinite loop. In PHANtom, the code however does not loop and solely prints Showing reentrancy control on the transcript. This is achieved by letting the computation that is intercepted by the aspect (*i.e.,* Transcript show: 'reentrancy control'; cr.) be contained by its own membrane, and the computation of the aspect be a separate membrane in which the aspect is registered. The aspects' membrane is then bound to the computations' membrane. Join points of the computations' membrane flow to the aspects' membrane, but join points of the aspects' membrane do not flow to itself. Hence the aspect does not capture its own computation and thus does not loop.

2.3.3. User-Specified Membrane Topologies PHANtom provides the programmer the power to define a custom topology of membranes and their binding relations, which aspects are registered in what membranes, and on what computation the membranes are deployed by means of pointcuts. The API of membranes is shown in Table IV. It allows the programmer

¶In Pharo, Transcript is actually an instance of a different class (that depends on the version of Pharo). Hence for the code to work the argument to receivers: needs to be changed to Transcript class asString

DOI: 10.1002/spe

to straightforwardly specify situations where join points emitted by aspects need to be visible to (other) aspects, in a structured manner.

The membranes paper [19] provides a number of examples of how such advanced topologies can be achieved through the use of membranes. We repeat here the example used to illustrate how a directed acyclic graph of membranes and their binding can be constructed. Suppose that we have a browser computation that is complemented by a cache aspect and a quota aspect. The cache aspect intercepts URL requests by the browser to cache the retrieved pages locally on disk. The quota aspect should verify that disk operations by the browser (*e.g.*saving a page to disk) as well as the cache do not exceed the allowed disk quota. The implementation of this topology is shown below:

```
|browserMem cacheMem quotaMem|
" assuming browser class, cache and quota aspects "

browserMem := PhMembrane new pointcut: (PhPointcut receivers: 'Browser' selectors: #()).
browserMem install.

cacheMem := PhMembrane new.
cache registerOn: cacheMem.
cacheMem advise: browserMem.

quotaMem := PhMembrane new.
quotaMem advise: browserMem; advise: cacheMem.
quota registerOn: quotaMem.
```

In this code we see three groups of statements. The first group creates a membrane, stating the computation around which it will be deployed using a pointcut, and deploys it. In the second group, the cache aspect is registered in a second membrane and this cache membrane is bound to the browser membrane. By registering an aspect in a membrane, the membrane is automatically deployed on the aspect, *i.e.,*the join points of the aspect are captured by the membrane. This will enable the quota aspect to match on join points of the cache aspect. In the third group, the quota aspect is registered in its own membrane as well, and the quota membrane is bound to both the browser and cache membrane.

3. PHANTOM USAGE PATTERNS

A significant part of the practice of using Smalltalk lies in its usage patterns, idioms and programmer conventions [2]. For example, Smalltalk has no visibility specifiers on methods, instead relying on categorizing methods in a protocol named private or public to indicate their private implementation or public interface. As PHANtom is a Smalltalk aspect language, a sizable part of the practice of using PHANtom also lies in the use of such patterns.

In this section we introduce two kinds of such patterns: First we show how we can combine PHANtom language features to make pointcuts public and aspects abstract, in the AspectJ sense. Second we present patterns for different kinds of aspect instantiation and deployment.

3.1. Public Pointcuts and Abstract Aspects

AspectJ provides several language features not present in PHANtom, but which are readily achieved through a combination of its constructs. As an example, we now show usage patterns that implement named pointcuts and abstract aspects. These examples also serve as an illustration of how the simplicity of PHANtom does not necessarily yield an inferior language, thanks to the power of some of its features.

In AspectJ, *named pointcut declarations* are similar to static variables: the pointcuts have a name and a visibility specifier, and can be accessed from outside of the aspect, as permitted by their visibility. In PHANtom we can provide named pointcuts by using a class method of the

DOI: 10.1002/spe

aspect that simply returns the corresponding pointcut instance. Hence to obtain a pointcut when needed, a message is sent to the relevant aspect class. A straightforward implementation of the class method would instantiate a new PhPointcut on each execution. For efficiency's sake however the pointcut may be kept as a class variable and be returned by the class method.

AspectJ *abstract aspects* are aspects that possess a pointcut that is declared to be abstract. Such aspects may be subclassed, yielding a concrete aspect when all abstract pointcuts are made concrete. Note that, in AspectJ, concrete aspects may not be subclassed. As PHANtom aspects are regular classes, they conform to the normal Smalltalk conventions for abstract classes and methods: A method is considered abstract if its implementation is the expression self subclassResponsibility, and classes are considered abstract if they contain abstract methods. PHANtom aspects therefore are abstract in the conventional sense if any of their methods are implemented in this fashion. This also automatically allows for abstract aspects in the AspectJ sense, as follows. Similar to the usage pattern above, the implementation of the aspect may obtain pointcuts by sending itself a message. If the implementation of the corresponding method consists of the self subclassResponsibility expression, the pointcut is considered as abstract, and hence the aspect is abstract. Subclasses of the abstract aspect simply override this method with a concrete implementation. Note that, by following Smalltalk conventions, PHANtom has no restrictions on subclassing aspects, *i.e.,* concrete aspects may also be subclassed, where AspectJ concrete aspects cannot.

3.2. Aspect Instantiation and Deployment Patterns

3.2.1. The Role of the Initialize Method In PHANtom, objects differ from aspects when considering their instantiation. In the elementary case of object instantiation, instantiating an object by sending its class the new message returns an instance that is fully formed. Aspects are however different because for them to be fully functional, advice and inter-type declarations need to be added to the newly created instance. The PHANtom convention to achieve fully-formed aspects as a result of the new message send has already been shown in Section 2.1.5. It consists of adding the advice and inter-type declarations in the initialize method. This method is called by the new method to initialize the object with default values, if needed. In PHANtom it thus acquires as extra responsibility the adding of the needed advice and inter-type declarations to self. Also, if it is expected that instantiating an aspect automatically activates it, this is also specified in initialize by sending the install message to self (again, as already shown in Section 2.1.5). As a result of this convention, aspect instantiation behaves identically to object instantiation, which aligns with the simplicity goal of PHANtom.

A more complex case of instantiation is when objects expect initial values for their instance variables at instantiation time. For this case, one pattern in Smalltalk is to implement a class method that instantiates the object and then fills in these variables. Typically it uses the values that were passed as arguments. Similarly, a PHANtom aspect may require some instance variables to contain relevant values for the pointcuts, advice or inter-type declarations that are added in the initialize method. In such cases a class method should be provided for instantiation. This method should instantiate the aspect using the basicNew method[‖], fill in the instance variables as required, and then call the initialize method on the instance. As a consequence, the responsibility of the initialize method does not differ from the elementary case to the more complex case, again aligning with the goal of simplicity of PHANtom.

3.2.2. Singleton Aspects In aspect languages such as AspectJ, by default aspects are automatically instantiated and only one instance of the aspect is present in the system. Such *singleton aspects* are straightforwardly implemented in PHANtom as well.

Implementing a singleton aspect in PHANtom is almost identical to implementing a singleton object in Smalltalk. In the standard Smalltalk fashion, the class of the aspect contains a variable

[‖]This method is equivalent to the new method except that it does not call the initialize method on the instance.

DOI: 10.1002/spe

that holds the unique instance, the new method is overridden to avoid instantiating more than one object, and a class method that provides a reference to the instance is provided. The only extra effort that needs to be performed is to ensure that there is always an instance present, even if the singleton aspect is never referenced. This explicit bootstrapping is required as the aspect itself defines where it applies in the system, in contrast to singleton objects that need to be referenced before they can be sent messages. Hence a class side initialize method needs to be defined that instantiates the aspect and places it in the class variable. As a consequence, whenever the class is loaded, the singleton aspect will be instantiated.

3.2.3. Anonymous and Deployment Aspects PHANtom aspects do not necessarily need to be instances of a subclass of PhAspect to provide useful behavior. Advice and inter-type declarations can be added to instances of PhAspect itself to yield a fully functional aspect. We call such aspects *anonymous aspects*. They are useful for succinctly stated behaviors, *e.g.* tests, brief examples (such as the example shown in Section 2.3) and deployment aspects.

Deployment aspects are aspects whose sole responsibility is to install and uninstall other aspects, such that the installed aspects are active only when needed, where needed. An example usage of such a deployment aspect is to achieve per-object aspects: for each object of a certain class that is instantiated, an accompanying aspect is instantiated. This aspect furthermore contains a reference to the object that it is associated with. The example below shows such a deployment aspect that produces a MyAspect instance for each instance of MyClass.

```
|depl|
depl := PhAspect new.
depl add: (PhAdvice around:
            (PhPointcut receivers: 'MyClass class' selectors: 'new' context: #(proceed))
                        advice: [:ctx| |obj| obj := ctx proceed. MyAspect assoc: obj. obj]).
depl install.
```

The pointcut of the above code intercepts the new message of MyClass class and exposes the proceed context element to the advice. The advice block first invokes the original behavior by calling this context element, holding on to the returned value, *i.e.,* the newly created object. Then it instantiates the MyAspect aspect by calling the assoc: class method, passing it the newly created object it is associated with. Lastly it returns the newly created object.

3.2.4. PerThis and PerTarget Aspects Deployment aspects, as above, allow us to implement the equivalent of the AspectJ aspect instantiation policies called *perThis* and *perTarget*. Both of these instantiation policies take as argument a pointcut, say ppc, and specify a per-object aspect instantiation policy, where the created aspect is associated to one object. The advice of the aspect will only run if the object that raised the join point is the associated object. The difference between both policies are the objects to which the aspect is associated. In the former these are the senders of messages producing the method executions identified by ppc, in the latter they are the receivers of these messages.

In PHANtom, the equivalent of such aspect instantiation for the pointcut ppc is achieved as follows. First, a per-object deployment aspect, as above, is used. It is responsible for instantiating one aspect for each object that may be sender, resp., reciever of the method identified by ppc. Second, in the deployed aspect all pointcut instantiations are suffixed by one of the two lines in the following code:

```
& (ppc if: [:ctx| ctx sender = self target]). "perThis"
& (ppc if: [:ctx| ctx receiver = self target]). "perTarget"
```

The former yields an expression that ensures that at runtime only the aspect that is associated to the sender executes its advice. The latter yields an expression that ensures that only the aspect that is associated to the receiver executes its advice.

DOI: 10.1002/spe

4. IMPLEMENTING SOFTWARE USING PHANTOM

We have used PHANtom to refactor fairly complex pieces of software, validating the usefulness of the language. (Note that while both these cases consist of a refactoring, this does not mean to imply that PHANtom can only be used this way.) We summarize two of these experiments here, the first example illustrating the ability to modularize functionality that cross-cuts extensively, and the second example showing the benefits of PHANtom over another mechanism to modularize a cross-cutting concern: Traits.

4.1. Refactoring Spy

Spy [3] is a framework for dynamic program monitoring in Pharo. It has been used to define execution time profilers, memory use profilers and test case coverage profilers, amongst others. An interesting property of Spy is that it can be used to generate visual reports, yielding a diagram of the profile. Example visual reports of execution time profiles are shown in Fig 4.

To perform monitoring, Spy instruments all classes being monitored such that the execution of their methods is prefixed and postfixed with program monitoring code. As a result, the monitoring functionality of Spy cuts across the entire application being monitored. As a case study for PHANtom, we successfully replaced this custom instrumentation with a number of aspects that provide the same functionality. This replacement is furthermore completely transparent for the users of Spy since the form in which concrete profilers are specified, *i.e.,* how to instantiate the framework, has not been changed.

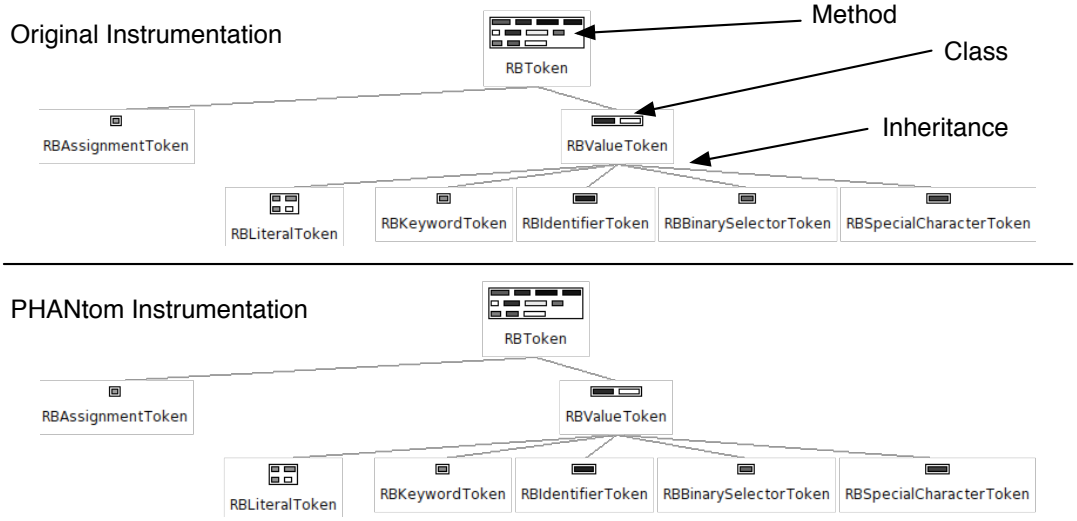

Figure 4. Comparison of original instrumentation results versus PHANtom instrumentation results. Example classes, methods and inheritance links are indicated in the figure. Method width is proportional to execution count, height to execution time and color to the number of receivers.

The implementation of the monitoring in PHANtom consists of three main parts: pointcut specification, per-instrumented-class aspects and per-profile membranes. First the pointcuts need to be constructed, based on the specification of the user of the framework. This specification solely contains classes. From it the corresponding pointcuts are created that jointly specify the interception of all the methods of these classes. Second, one aspect instance is created of the SpyAspect class for each class that is monitored. These SpyAspect instances contain one advice for each method of the class. This advice is responsible for performing the program monitoring and calling the original method behavior. Third, to minimize the overhead of membranes only one membrane is created to envolve the entire computation that is being

DOI: 10.1002/spe

profiled. All the aspects register themselves in that membrane. Otherwise each aspect would create its own membrane, yielding a large number of membranes where one suffices.

We compared the original instrumentation of Spy to the PHANtom instrumentation by running both versions of Spy on example cases. As a representative result, we show here the results of running all the tests present in RBFormatterTest (testing code formatting in the Refactoring browser). Figure 4 shows the results of the execution time profiler, for both versions. The figure shows that both profiles are nearly identical, with negligible differences in execution time of methods. Considering memory usage profiling, the returned results are similar but do not match as well as Fig. 4. This is as the overhead of the PHANtom infrastructure in terms of memory usage is different to the original instrumentation code. We do not consider this as a serious issue, as each infrastructure used for monitoring will have a specific memory usage pattern. Hence such an artefact will be always present in the collected results.

4.2. Refactoring AspectMaps

AspectMaps [10] is a visualization whose purpose is to facilitate code comprehension of Aspect-Oriented programs. In this section we detail a number of refactorings made on the tool, and hence only focus on relevant parts of its implementation. For more details we refer to [10].

In AspectMaps, the code to be visualized is reified as a Moose [7] model. This is a hierarchical model that contains the relevant entities for the visualization such as packages, classes, aspects, methods and advice, along with the relevant cross-cutting relationships. The implementation of the visualization uses the Mondrian [14] information visualization engine. To visualize the program, the model is traversed and each entity in the model informs Mondrian as to how it should be visualized.

Visualization of entities of the model is therefore a cross-cutting concern: the code of visualization is scattered among all the classes that represent model entities. AspectMaps is remarkable that it already uses Traits [8] in an attempt to modularize this concern. In summary, Traits are partial class descriptions that only contain the definition of methods. They can be composed with existing classes, adding their methods to the class. This however does not allow for state to be added in a cross-cutting fashion, and neither do traits enable implicit invocation. This resulted in an incomplete modularization of the AspectMaps tool.

With PHANtom we were however able to obtain a modularization that overcomes the above limitations and we discuss this next.

4.2.1. Cross-cutting State Addition In AspectMaps, packages, classes, aspects and methods are visualized in two possible ways: either in a compact form or in an extended form. Each package, class, aspect and method instance has a variable whose contents determine whether the compact or the extended form of that entity should be shown. This requires adding the instance variable to the class definition of all four classes of these entities, as they have no common direct parent in the class hierarchy.

Traits cannot be used to solve this problem of cross-cutting state addition, since traits (as present in Pharo) do not define state. We hence defined a PHANtom aspect to add this state and its accessors using straightforward inter-type declarations, as follows:

```
AMVisState>>initialize
  | pc |
  super initialize.
  pc := PhPointcut receivers: #(ASPIXAspect FAMIXClass FAMIXMethod FAMIXNamespace)
     selectors: #().
  self addClassModifier:(PhClassModifier on: pc addIV: 'shape').
  self addClassModifier:(PhClassModifier on: pc addIM: 'extShape\shape:=''ex''' withCRs).
  self addClassModifier:(PhClassModifier on: pc addIM: 'compShape\shape:=''cp''' withCRs).
  self addClassModifier:(PhClassModifier on: pc addIM: 'isCompShape\↑shape=''cp''' withCRs).
  self install.
```


DOI: 10.1002/spe

4.2.2. Modularizing User Interaction The AspectMaps visualization is interactive: clicking on the visualization of classes, methods and aspects causes their visualization to be toggled between the compact and the extended shape. For this to occur a piece of code is repeated six times: in each of these three classes it is present in the beginning of both methods that draw compact and extended visualization. To modularize this it is however not sufficient to extract this piece of code to one method that is placed in a trait. As the method still needs to be invoked from the place where it was extracted, we are again faced with code that is repeated six times. In contrast, by extracting this piece of code to an advice we are able to completely modularize it, since the aspect specifies the advice invocation, as shown below:

```
AMInteractionLogic>>initialize
  |curshape excont|
  super initialize.
  curshape := PhPointcut receivers: #(ASPIXAspect FAMIXMethod FAMIXClass)
    selectors: #currentShapeFor:withProperties: context: #(arguments receiver).
  excont := PhPointcut receivers: #(ASPIXAspect FAMIXMethod FAMIXClass)
    selectors: #extendedContentsFor:withProperties: context: #(arguments receiver).
  self add: (PhAdvice before: curshape send: #drawingPreliminaries: to: self).
  self add: (PhAdvice before: excont send: #drawingPreliminaries: to: self).
  self install.
```

5. RELATED WORK

Beyond being the first work on AOP, AspectJ [13] is the reference work for aspect languages. AspectJ is an AOP extension to Java, allowing aspects to be woven at compile time or at class loading time, hence it has no dynamic features. In AspectJ, an aspect can be seen as a class that cannot be instantiated and which may also contain pointcuts and advice. Pointcuts are defined in a specific DSL that allows for patterns to specify class and method name, as well as the type of pointcut: method call, object instantiation, field access, and so on. Advice specifications are similar to methods, named before, after or around, indicating before, after or around advice. Context exposed by the pointcut is made available as arguments to the advice and the pointcut identification is placed before the method body. Lastly, inter-type declarations are field or method declarations whose signature is prepended by the target type name and a dot.

AspectS [12] is a seminal work on aspect-oriented programming, presenting an AOP extension to Squeak Smalltalk. It is the first aspect language to feature dynamic AOP, *i.e.,* the addition and removal of aspects at runtime. AspectS has no dedicated pointcut language, instead relying on the use of metaprogramming to specify the join point shadows, and what are termed *advice qualifiers* to perform runtime testing of the join points. The behavior of advice is given by using a block, and the context is passed to the block as multiple arguments. The implementation of AspectS combines method wrappers [4] with metaprogramming. The former are used to intercept computation at join point shadows, while the latter is used in various places, *e.g.* to determine join point shadows or to obtain the activation context of the message sender. AspectS also provides various extensions of the Squeak browsers to support, *e.g.* navigation between join point shadows and aspects, and vice-versa. AspectS however does not provide explicit support for advice ordering, nor explicit support for controlling reentrancy.

Another new aspect language for Smalltalk is PHASE [15]. The goal of PHASE is however radically different from PHANTom: it is designed and implemented as a validation of the concept of *meta join point model*. This model defines new join points to characterize the structure and behavior of aspects, together with the corresponding pointcut language predicates. The ultimate goal of the work is to use aspects to compose aspects, since aspect composition can be considered a cross-cutting concern. PHANtom does not have a meta join point model, and instead relies on the advice ordering features outlined in Sect. 2.2. There are

DOI: 10.1002/spe

various similarities between both languages, we highlight the following: all constructs are first class, patterns can be used in pointcuts, dynamic pointcuts are supported as well as inter-type declarations and the latter use a pointcut to determine where they apply. Notable differences (beyond the meta join point model) are first and foremost the complexity of PHASE versus the simplicity of PHANtom. Feature-wise we distinguish that PHASE has a richer join point model; also including the sending of messages and accesses to instance variables, PHASE advice are only of the around kind and do not have their behavior as a separate method, aspects can only inherit from abstract aspects and are implicitly instantiated. Also, to the best of our knowledge, PHASE has only been used as a proof of concept, and not in the implementation of non-trivial applications.

Eos-U [16] is an aspect language for C#, in the style of AspectJ, that unifies aspects and objects into a more symmetrical view of AOP. Initially, in AspectJ, aspects were seen as entities fundamentally different from normal classes. In AspectJ, Aspects could not be (and still cannot be) instantiated, and join points in the execution of advice could not be captured by other aspects. Eos-U introduced the concept of *classpects*, which remove these separations between classes and aspects. Classpects can be instantiated and their behavior *i.e.,* their advice, is implemented as regular methods, whose join points may be seen by other classpects. Classpects may contain AspectJ-like pointcut specifications that now also are responsible for identifying the associated advice and its type (before, after or around). The latter however restricts the reuse possibilities of pointcuts and complicates logical combinations of pointcuts. This is why in PHANtom the advice construct is still present and represents the binding between a pointcuts and the behavior to be executed.

PHANtom also builds on the work by Assaf and Noyé on Dynamic AspectJ, which has been discussed in Section 2.2.2, and work by Tanter *et al.*, which has been talked about in Section 2.3. We do not further go in detail on this work here but instead refer to the respective sections.

6. CONCLUSIONS AND FUTURE WORK

We have introduced PHANtom, a modern aspect language for Pharo Smalltalk. PHANtom is an aspect language in the spirit of Smalltalk, combining dynamism, simplicity and power. PHANtom is a modern aspect language as it is inspired by features of existing aspect languages, integrates recent research results and is created with optimization in mind. PHANtom is built, on the one hand, as a base for experimentation of aspect-oriented programming in Smalltalk, and on the other hand, intends to provide a modern aspect language to the Smalltalk programmer.

In this paper, we first gave an overview of PHANtom, starting with the design considerations and detailing what features are taken from which previous work. We then described how pointcuts, advice, inter-type declarations and aspects are created, and how the latter are deployed. We followed this with a discussion of the more advanced features of PHANtom: aspect ordering at deployment time and at runtime, and management of reentrancy using computational membranes.

After the language introduction, we presented usage patterns of PHANtom. Similar to the high significance of usage patterns in Smalltalk, usage patterns in PHANtom form a significant part of the practice of using PHANtom. Consequently we detailed relevant usage patterns, *e.g.* showing how language functionality of AspectJ is realized in PHANtom using patterns. This was followed by an illustration of two uses of the language to modularize cross-cutting concerns in fairly complex applications. First we have shown how a highly cross-cutting concern of Spy [3] was modularized, and second we have shown how PHANtom outperforms Traits [8] in the implementation of AspectMaps [10].

A first avenue of future work is optimization of the existing implementation. A major part of this is compiling the generation of join points and invocation of advice, as this is currently performed through meta-level operations. Second is the extension of the join point model to

DOI: 10.1002/spe

also include instance variable access. Next to this, we are considering to develop a specific syntax for PHANtom, to be able to program aspects in a more concise manner. Lastly, as tool support for PHANtom is currently lacking, we plan to add tool support that allows to see the impact of aspects in the code browser, or even use specific aspect visualizations such as AspectMaps.

ACKNOWLEDGEMENTS

We thank Éric Tanter and Romain Robbes for their invaluable comments on draft versions of this paper. We are also grateful to Éric for permitting us to take a figure from the computational membranes paper (Figure 3 in this text).

References

1. Ali Assaf and Jacques Noyé. Dynamic AspectJ. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 8:1–8:12, New York, NY, USA, 2008. ACM.
2. Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1996.
3. Alexandre Bergel, Felipe Banados, Romain Robbes, and David Roethlisberger. Spy: A flexible code profiling framework. *Computer Languages, Systems and Structures*, 38(1):16 – 28, 2012.
4. John Brant, Brian Foote, Ralph Johnson, and Donald Roberts. Wrappers to the rescue. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 396–417. Springer Berlin / Heidelberg, 1998.
5. Ruzanna Chitchyan, Johan Fabry, Shmuel Katz, and Arend Rensink. Editorial for special section on dependencies and interactions with aspects. *Transactions on Aspect-Oriented Software Development*, 5490:133–134, 2009.
6. Thomas Cleenewerck, Johan Fabry, Anne-Francoise Lemeur, Jacques Noyé, and Éric Tanter, editors. *Proceedings of the 4th workshop on Domain-Specific Aspect Languages*, Charlottesville, VA, USA, March 2009. ACM Press.
7. Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.
8. Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.
9. Johan Fabry and Daniel Galdames. PHANtom: a modern aspect language for Pharo Smalltalk. In *Proceedings of IWST2011*. ACM Press, 2011. To Appear.
10. Johan Fabry, Andy Kellens, and Stéphane Ducasse. AspectMaps: A scalable visualization of join point shadows. In *Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC2011)*, pages 121–130. IEEE Computer Society Press, Jul 2011.
11. Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 26–35, New York, NY, USA, 2004. ACM.
12. Robert Hirschfeld. AspectS - Aspect-Oriented Programming with Squeak. In Mehmet Aksit, Mira Mezini, and Rainer Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer Berlin / Heidelberg, 2003.
13. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer Berlin / Heidelberg, 2001.
14. Adrian Lienhard, Adrian Kuhn, and Orla Greevy. Rapid prototyping of visualizations using Mondrian. In *Proceedings IEEE International Workshop on Visualizing Software for Understanding (Vissoft'07)*, pages 67–70, Los Alamitos, CA, USA, June 2007. IEEE Computer Society.
15. Antoine Marot. *Preserving the Separation of Concerns while Composing Aspects with Reflective AOP*. PhD thesis, Université Libre de Bruxelles, 2011.
16. H. Rajan and K.J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 59 – 68, may 2005.
17. Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010.
18. Éric Tanter. Execution levels for aspect-oriented programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 37–48, New York, NY, USA, 2010. ACM.
19. Éric Tanter, Nicolas Tabareau, and Rémi Douence. Taming aspects with membranes. In *Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012)*, Potsdam, Germany, March 2012. ACM Press. To appear.

DOI: 10.1002/spe